C++ std::string

Daniel Plakosh, Software Engineering Institute [vita[1]]

VERBETER

This article was marked with the Pearson Education copyright footnote and a CMU copyright both. Check with author. Fix the Footer part as well.

2005-09-26

C++ programmers have the option of using the standard std::string class defined in ISO/IEC 14882. The std::string generally protects against buffer overflow.

Development Context

String manipulation

Technology Context

C++, UNIX, Win32

Attacks

Attacker executes arbitrary code on machine with permissions of compromised process or changes the behavior of the program.

Risk

Standard C string manipulation functions are prone to programmer mistakes that can result in buffer overflow vulnerabilities.

Description

Unbounded string copies are not limited to the C programming language. For example, if a user inputs more than 11 characters into the C++ program shown in Figure 1, an out-of-bounds write will result.

Figure 1. Vulnerable program combining C and C++ standard strings

```
#include <iostream.h>
int main() {
  char buf[12];
  cin >> buf;
  cout << "echo: " << buf << endl;
}
```

The standard object cin is an instantiation of the istream class. The istream class provides member

1. daisy:268 (Plakosh, Daniel)

functions to assist in reading and interpreting input from a stream buffer. All formatted input is performed using the extraction operator operator>>. C++ also defines external operator>> overloaded functions that are global functions and not members of istream, including

```
istream& operator>> (istream& is, char* str);
```

This operator extracts characters and stores them in successive locations starting at the location pointed to by str. Extraction ends when the next element is either a valid white space or a null character, or if the end-of-file is reached. A null character is automatically appended after the extracted characters.

The extraction operation can be limited to a specified number of characters (thereby avoiding the possibility of an out-of-bounds write) if the *field width* inherited member (ios_base::width) is set to a value greater than 0. In this case, the extraction ends one character before the count of characters extracted reaches the value of the field width, leaving space for the ending null character. After a call to this extraction operation, the value of the field width is reset to 0.

Figure 2 contains an improved version of the Figure 1 program that sets the field width member to the length of the character array.

Figure 2. Extracting characters using the field width member

```
#include <iostream.h>
int main() {
  char buf[12];
  cin.width(12);
  cin >> buf;
  cout << "echo: " << buf << endl;
}
```

While setting the field width solves the buffer overflow problem, it does not address the issue of truncation. Therefore, unexpected program behavior could result when the maximum field width is reached and the remainder of characters in the input stream are consumed by the *next* call to the extractor operator.

C++ programmers have the option of using the standard std::string class defined in ISO/IEC 14882 [ISO/IEC 98]. The std::string class is the char instantiation of the std::basic_string template class, and it uses a dynamic approach to strings in that memory is allocated as required—meaning that in all cases, size() <= capacity(). The std::string class is convenient because the language supports the class directly. Also, many existing libraries already use this class, which simplifies integration.

Figure 3 shows another solution to extracting characters from cin into a string, using std::string instead of a character array. This program is simple, elegant, handles buffer overflows and string truncation, and behaves in a predictable fashion.

Figure 3. Extracting characters from cin into a std::string object

```
#include <iostream>
#include <string>
using namespace std;
int main() {
  string str;
  cin >> str;
  cout << "str 1: " << str << endl;
}
```

The std::string generally protects against buffer overflow, but there are still situations in which programming errors can lead to buffer overflows. While C++ generally throws an out_of_range exception when an operation references memory outside the bounds of the string, the subscript operator [] (which does not perform bounds checking) does not [Viega 03].

Another problem occurs when converting std::string objects to C-style strings. If you use string::c_str() to do the conversion, you get a properly null-terminated C-style string. However, if you use string::data(), which writes the string directly into an array (returning a pointer to the array), you get a buffer that is not null terminated. The only difference between c_str() and data() is that c_str() adds a trailing null byte.

Finally, many existing C++ programs and libraries have their own string classes. To use these libraries, you may have to use these string types or constantly convert back and forth. Such libraries are of varying quality when it comes to security. It is generally best to use the standard library (when possible) or to understand the semantics of the selected library. Generally speaking, libraries should be evaluated based on how easy or complex they are to use, the type of errors that can be made, how easy these errors are to make, and what the potential consequences may be.

References

[ISO/IEC 99]	ISO/IEC. *ISO/IEC 9899 Second edition 1999-12-01 Programming languages — C*. International Organization for Standardization, 1999.
[ISO/IEC 98]	Joint Technical Committee ISO/IEC JTCI; International Organization for Standardization; and International Electrotechnical Commission. *Programming Languages — C++*. Geneva, Switzerland: ISO/IEC, 1998.
[Viega 03]	Viega, John & Messier, Matt. *Secure Programming Cookbook for C and C++: Recipes for Cryptography, Authentication, Networking, Input Validation & More*. Sebastopol, CA: O'Reilly, 2003 (ISBN: 0-596-00394-3).

Velden

Naam	Waarde
is-content-area-overview	false
Content Areas	Knowledge/Coding Practices
SDLC Relevance	Implementation
Workflow State	Publishable